Data Movement with MPI in a Multi-Threaded World

Ryan E. Grant
Senior Member of Technical Staff
Sandia National Laboratories
Where do threads come from?

- MPI use cases continue to evolve
  - MPI+X implies the use of threads, e.g. OpenMP

- Potentially thousands of MPI processes on a single node (without threading)
  - Can complicate network resource management

- Sources of concurrency
  - Many core architectures
    - KNL
  - Increasing core counts
    - Traditional CPU design continues to add cores in new generations
  - OpenMP
    - Can we use MPI inside of OpenMP parallel regions?
  - New runtimes
    - Tasking runtimes potentially introduce many more concurrent uses of the MPI library
Living in a World with Threads

Desirable/required features:

- Low overhead
- Similar semantics to existing threading (minimal changes)
  - Ease of programmability
- Each thread has access to the communication library (no funneling)
  - Also ease programmability, e.g. use MPI calls in an OpenMP region
  - Preferably keep programmability similar to today’s
- Communication endpoint granularity matched to the work
  - Not too fine, not too coarse, just right...
  - Fine granularity at endpoints requires networking resources
    - Keeping track of many ranks, caching state related to these ranks, etc.
The problem with threads

- Threads introduce significant issues with concurrency in existing MPI implementations
  - MPI_THREAD_MULTIPLE is hard and implementations don’t do it well
  - Difficult to support concurrency without encountering conflicts

- Concurrency exists at multiple levels
  - Inside applications, where threads are used
  - MPI implementations, where threads can be used independent from application threads

- Different types of threads exist
  - OS level threads (POSIX Pthreads)
  - User level threads (e.g. Qthreads, Argobots)
New Architectures + Threads

- Message rate is key to communication
  - New architectures are leading to lower message rates per core
  - Need to use multiple cores to achieve higher message rate

Note that big cores are an order of magnitude better at message rate

New Architectures + Threads

- Potential solution -> Use more ranks (processes)

16 ranks gets us to POWER7 matching message rates, or ~1/2 of Haswell rates

But ranks require communication resources, can we use threads instead and share ranks?

Multiple ranks running on a Intel Xeon Phi (KNC-B0)

New Architectures + Threads

- Multiple threads is much less impressive than multiple processes

![Graph showing impact of number of threads on message rate and bandwidth](image)

Need 4 procs with 4 threads each to exceed 1 proc in message rate!

---

New Architectures

- Decreasing core frequency also impacts message rate
  - Depends on how much HW support you have to help you

New Architectures

- Even with hardware support, and similar frequency many-cores are still dominated by deep out-of-order CPUs

![Graphs showing message rate vs. queue depth for Sandy Bridge and Xeon Phi processors at various frequencies.](image)

New Architectures + Threads

- Multiple ranks might get us some of the way to utilizing new architectures as efficiently as current big cores
  - But require communication resources for each rank
  - Large rank tables consume lots of memory
  - Large rank space requires longer bootstrapping times at initialization

- MPI needs more efficient ways to handle threads
  - Question: Do we need per thread addressability or not?
  - It can be hard to know which thread on the target node is actually the one that needs the data
  - Many-task schemes make this very hard/impossible to know
  - Programming is easier if data management is done on the receive side
Evolving MPI

- Fixing MPI concurrency issues is hard, so multiple approaches have been proposed:
  1) Parallelize critical paths inside MPI implementations
     - Make the fast path faster by allowing concurrency
     - Match list parallelization or new matching methods
     - Parallelize linked list search or move over to hashing approaches
  2) Expose the comm. parallelism to applications (Endpoints)
     - Allow individual threads to have ranks in MPI (rank space explosion)
     - Exposes lots of complexity, thread addressability on the network
     - General solution that is applicable to many problems
  3) Allow for parallelism in a way that doesn’t require MPI to handle it
     - Threads can contribute independently to a larger communication operation inside of MPI, doesn’t require the same synchronization methods
MPI Evolution #1

- Added Concurrency/increased efficiency implementation
- Message matching efficiency is greatly improved
  - Incorporate multi-threaded methods of matching
    - Tricky, but possible, and improves message rate
  - Move away from traditional linked lists
    - Hashing approaches (Intel Mitigating MPI Matching Misery (ISC))
    - Hybrid hashing w/ lists (Purdue)
    - Alternative approaches for quick lookup (SNL/Queen’s University)
    - Hardware solutions (Sandia, Bull-Atos)
      - FPGAs, ASIC design (TCAM), hardware level concurrency (many cores)
  - Hardware to make pointer chasing better
    - Using vector units to fetching multiple match entries at a time (SNL)
    - Staging match data so that it is close to CPUs (SNL)
MPI Evolution #1

- Reducing the amount of locking necessary
  - Enhance MPI with fine grained locks
    - We know this is possible/useful from old commercial MPIs
  - Providing separate distinct networking contexts to threads can help reduce locking frequency
  - Radical changes to account for threads like FG-MPI (UBC)

- Adopt runtimes that do the work for us
  - Tasking runtimes can help in ordering requests to prevent locking at the MPI level
MPI Evolution #2

- Exposing communication parallelism to applications
  - MPI Endpoints proposes assigning ranks to threads
    - Not necessarily 1-to-1 mapping
  - Creates a new communicator with expanded rank space
    - This can be problematic as it requires more resources
    - It also causes issues with hardware virtual->physical rank mapping
      - Rank space did not typically grow before
  - Requires knowledge of target thread on receiver side
    - Or groups of target threads for non 1-to-1 mapping
  - Rank scalability is not much better than MPI everywhere
    - Sparse mapping can help with this (on-demand/dynamic)

- Endpoints *may* be a solution for general cases, but they have some issues
MPI Evolution #3

- Handle concurrency in a way that MPI doesn’t have to
  - Several possible solutions in this space like runtimes, or new MPI functionality

- Runtimes can manage parallel communication requests
  - Like Intel circular buffer with a dedicated thread (software offload)
    - Increases message rate over MPI_THREAD_MULTIPLE
  - User level threading runtimes can also help
    - MPIQ – qthreads runtime that serializes MPI requests

- MPI can evolve in a way that better supports concurrency
  - SNL/AuburnU/UAB solution to handle most use cases - finepoints
Finepoints-Better Parallelism in MPI

- Concept of many actors (threads) contributing to a larger operation in MPI
- E.G. many threads work together to assemble a message
  - MPI only has to manage knowing when completion happens
  - These are actor/action counts, not thread level collectives, to better enable tasking models
- No heavy MPI thread concurrency handling required
  - Leave the placement/management of the data to the user
  - Knowledge required: number of workers, which is easily available
- Match well with Portals NIC capabilities
  - Use counters for sending/receiving
  - Utilize triggered operations to offload sends to the NIC
Persistent Partitioned Buffers

- Expose the “ownership” of a buffer as a shared to MPI
- Need to describe the operation to be performed before contributing buffer segments
- MPI implementation doesn’t have to care about sharing
  - Only needs to understand how many times it will be called before the operation can be completed
- Threads are required to manage their own buffer ownership such that the buffer is valid
  - The same as would be done today for code that has many threads working on a dataset (that’s not a reduction)
- Result: MPI is thread agnostic with a minimal synchronization overhead (atomic_inc)
  - Can alternatively use task model instead of threads
Example

- Like persistent communications, setup the operation
  
  MPI_Partitioned_Send_Create(Comm, to_rank, to_tag, base_address, data_type, count, num_contributors, &request)

- Start the request
  
  MPI_Start(request)

- Add items to the buffer
  
  #omp parallel for ...
  
  MPI_Partitioned_Add_to_buffer(MPI_Request *request, const void *in_data_loc, int count, MPI_Datatype type)

- Wait on completion
  
  MPI_Wait(request)

- Optional: Use the same partitioned send over again
  
  MPI_Start(request)
Opportunities for Optimization

- MPI implementations can optimize data transfer under the covers
- Subdivide larger buffers and send data when ready
- Could be optimized to specific networks (MTU size)
- Number of messages will be:
  1 < #messages ≤ #threads/tasks
  For a partition with 1 part per thread
- Reduces the total number of messages sent, decreasing matching overheads
Different Approach to Threading

- Partitioned buffer operations can always be considered as multi-threaded
- Using partitioned sends doesn’t necessarily require locking in other parts of the MPI library
- Technically, using partitioned buffers could work with fast MPI thread modes
  - Violates the definition of thread modes in the standard
  - If only using partitioned sends, no need for locking in the library
- Not part of this work but...
  - Could examine alternative threading modes for MPI, where user management of data is guaranteed and only inherently thread safe operations are called
Finepoints Status

- Open MPI prototype is complete and debugged
- Performance results will be coming in the near future
- Optimization efforts will follow to demonstrate usefulness
- Re-write a mini-app to demonstrate how this approach would work
  - Also how much effort is required to change the code to support threads via this method
- Forum Proposal – dependent on testing/optimization results
Summary

- Threads are coming
  - Improvements to MPI are needed to maintain/improve performance

- Multiple approaches are needed for high probability of success

- Improving MPI under the covers may not be sufficient

- Exposing communication parallelism can be done in several ways
  - Need to be concerned with application change burden
  - More alternatives than discussed here
Acknowledgments:

This work was funded through the Computational Systems and Software Environment sub-program of the Advanced Simulation and Computing Program funded by the National Nuclear Security Administration